Все базовые операторы и функции выделения/освобождения памяти, строго говоря, являются небезопасными:

new, delete, malloc(), calloc(), realloc(), free()

Наиболее часто, в практике программирования, можно столкнуться со следующими ситуациями:

- память была выделена, но не освобождена (утечка памяти);
- память была освобождена, но работа с ней продолжается так, словно она остается выделенной;
- память не была выделена, но в нее выполняется запись данных;
- попытка несколько раз освободить одну и ту же область памяти.

К сожалению, даже при строгом контроле порядка вызовов операторов или функций выделения и освобождения памяти, вероятность возникновения приведенных ситуаций остается довольно высокой. Спрашивается, как минимизировать подобные ошибки, а еще лучше совсем их избежать? Для этого в языках высокого уровня, таких как Java, Python, C#, предусмотрен механизм «*сборки мусора*», который отслеживает неиспользуемую память и автоматически ее освобождает. Но в языке C++ нет такого сборщика мусора, т.к. это низкоуровневый язык программирования и сами сборщики пишутся зачастую на нем. Однако некоторые решения все же есть. Это использование, так называемых, умных (smart) указателей. *Они на примитивном уровне подобны сборщику мусора и сами контролируют процесс освобождения памяти*. То есть, вручную освобождать память уже не требуется, она освободится сама (автоматически), когда станет ненужной.

На этом и следующем занятиях мы как раз познакомимся с двумя видами smart-указателей:

- unique_ptr – «уникальный» (в единственном числе) smart-указатель, ссылающийся на выделенную область памяти (или принимающий значение nullptr);
- shared_ptr – smart-указатель, допускающий множественные адресации на одну и ту же выделенную область памяти.

Начнем с первого типа указателя unique_ptr.

Для использования в программе на языке C++ «умных» указателей необходимо вначале подключить заголовок:

```
#include <memory>
```

После этого в пространстве имен std появится класс unique_ptr. Теперь мы можем объявлять smart-указатели этого типа следующим образом:

```
std::unique_ptr<int> ptr;
std::unique_ptr<int> ptr_2 {};
std::unique_ptr<int> ptr_3 {nullptr};
```

Здесь сразу бросаются в глаза угловые скобки после класса unique_ptr. В них мы должны прописать тот тип данных, с которым будет работать указатель. В данном случае – это примитивный тип int. Вообще, забегая вперед, отмечу, что угловые скобки, стоящие после имени класса, – это пример использования шаблонного класса. То есть, *класс unique_ptr определен так, чтобы его можно было использовать с произвольными типами данных*. И благодаря этому достигается универсальность объявления smart-указателей типа unique_ptr. На данный момент достаточно просто запомнить, что после имени класса нужно прописать угловые скобки с типом данных, с которым в дальнейшем будет работать smart-указатель.

Итак, во всех трех приведенных вариантах объявления указателей, их значения равны nullptr, то есть, не ссылаются ни на какую выделенную область памяти. Поэтому, если сейчас попытаться вывести значение данных, на которые ссылается указатель ptr:

```
std::cout << *ptr << std::endl;
```

то программа завершится аварийно. Нельзя в языке C++ читать данные через указатель со значением nullptr. Правильнее вначале было бы проверить, что значение указателя не равно nullptr, то есть, он ссылается на выделенную область памяти. Это можно сделать следующим образом:

```
if(ptr)
    std::cout << *ptr << std::endl;
```

Обратите внимание, как записана проверка: nullptr – это эквивалент false, а не nullptr – это эквивалент true. Также обратите внимание, что чтение данных из памяти, на которую ссылается smart-указатель, синтаксически осуществляется так же, как и для обычного указателя.

Давайте теперь пропишем обратную проверку и если указатель ptr равен nullptr, то выделим для него область памяти. Начиная со стандарта C++14, делается это следующим

образом:

```
    if(!ptr)
        ptr = std::make_unique<int>(10);

    std::cout << *ptr << std::endl;
```

То есть, вызывается специальная шаблонная функция make_unique, в угловых скобках прописывается тип данных, под который выделяется память, а в качестве аргумента передается начальное значение 10. Теперь программа отработает без каких-либо проблем и в консоль будет выведено число 10.

Используя «умный» указатель, можно изменить значение данных в области памяти, на которую он ссылается:

```
*ptr = -7;
```

То есть, работа со smart-указателями осуществляется так же, как и с обычными указателями языка C++.

А что будет, если мы попробуем еще раз для этого же указателя выделить память с помощью функции make_unique:

```
ptr = std::make_unique<int>(10);
ptr = std::make_unique<int>(11);
```

Возникнет ли здесь «утечка памяти»? В действительности, нет, не возникнет. В классе unique_ptr операция присваивания переопределена и в момент присвоения нового адреса, прежняя область автоматически освобождается. Поэтому указатель ptr будет вести на новую область со значением 11.

Вообще, в практике программирования на C++, «умные» указатели довольно часто инициализируются сразу в момент объявления. Например:

```
std::unique_ptr<int> ptr {std::make_unique<int>()};
```

Причем, до стандарта C++14 эта инициализация прописывалась с помощью стандартного оператора new следующим образом:

```
std::unique_ptr<int> ptr_2 {new int {-6}};
```

Но теперь так делать не рекомендуется, хотя синтаксически и допустимо. Как минимум, по двум причинам:

1. Во-первых, области памяти, выделяемые под сам указатель ptr_2 и объект данных (int), могут располагаться независимо друг от друга. Тогда как по стандарту функции make_unique() рекомендуется располагать область данных smart-указателя и область данных, на которые он ссылается, непосредственно друг за другом. Это несколько ускоряет работу smart-указателя.
2. Во-вторых, при создании smart-указателя в аргументах какой-либо функции, оператор new и создание самого указателя могут происходить в разные моменты времени. В некоторых случаях такое рассогласование может приводить к непредвиденным ошибкам.

Итак, запомним, что выделение памяти для smart-указателей типа unique_ptr следует выполнять с помощью функции make_unique().

Далее, если у нас имеется два указателя типа unique_ptr, то мы не можем присваивать один другому. Например, команда:

```
ptr_3 = ptr_2;
```

приведет к ошибке на этапе компиляции. Это связано с тем, что указатель типа unique_ptr может ссылаться на ту или иную область памяти только в единственном числе. И никакой другой smart-указатель инициализировать или присвоить на эту же область уже не получится. В этом смысл «уникальности» smart-указателей этого типа. Хотя, «обмануть» компилятор все же возможно, если использовать такую конструкцию:

```
int* p = new int(5);
std::unique_ptr<int> ptr_2{ p };
std::unique_ptr<int> ptr_3{ p };
```

Программа скомпилируется, но может завершиться аварийно, так как оба smart-указателя ведут на одну и ту же область и дважды освобождают ее, что, в общем случае, недопустимо. Поэтому smart-указатели все же не панацея от всех возможных программных вариаций и программист должен грамотно и ответственно подходить к их использованию.

Smart-указатели типа unique_ptr имеют ряд полезных методов:

- get() – возвращает «сырой» указатель на выделенную область памяти;
- release() – возвращает указатель на выделенную область памяти и «отвязывает» smart-указатель от нее;

- reset() – меняет значение указателя на другую область памяти, либо на значение nullptr, если ничего не указано;
- swap() – выполняет обмен адресами smart-указателей между собой.

Давайте посмотрим на их работу. Первый метод get() достаточно прост. Он возвращает обычный (сырой) указатель, если в программе он требуется по каким-либо причинам:

```
int* p = ptr.get();
std::cout << p << " " << *p << std::endl;
```

Следующий метод release также возвращает сырой указатель на выделенную область, но сам smart-указатель «отвязывает» от этой области:

```
int* p = ptr.release();
std::cout << ptr.get() << " " << p << " " << *p << std::endl;
```

Если нам нужно изменить ресурс, на который ссылается «умный» указатель, то это делается с помощью метода reset:

```
ptr.reset(); // меняем на значение nullptr
```

или

```
ptr.reset(new int {143}); // меняем на новую область памяти
std::cout << ptr.get() << " " << *ptr << std::endl;
```

Обратите внимание, что здесь допустимо использовать оператор new, но не функцию make_unique() в чистом виде.

Наконец, последний метод swap() просто меняет адреса двух smart-указателей:

```
ptr.swap(ptr_2);
std::cout << *ptr << " " << *ptr_2 << std::endl;
```

Указатели типа unique_ptr на массивы

В заключение этого занятия отмечу, что указатели unique_ptr можно определять и на массивы. Например:

```cpp
#include <iostream>
#include <memory>

using std::cout;
using std::endl;

int main(void)
{
    unsigned total {10};
    std::unique_ptr<int[]> ar {std::make_unique<int[]>(total)};
    auto ar_2 {std::make_unique<int[]>(7)};
    std::unique_ptr<int[]> ar_3 {nullptr};

    return 0;
}
```

Затем, через ar, ar_2, ar_3 мы можем обращаться к элементам соответствующих массивов привычным нам образом:

```cpp
    for(int i = 0; i < total; ++i)
        cout << ar[i] << " ";
```

Увидим все нули. Это говорит о том, что при создании массивов с помощью функции make_unique() значения всех элементов автоматически инициализируются нулями.

Соответственно, можем записать какие-либо другие значения в массив следующим образом:

```cpp
    for(int i = 0; i < total; ++i)
        ar[i] = i*i;
```

То есть, работа выполняется абсолютно так же, как и с обычными массивами языка C++.